BUENAS PRÁCTICAS PARA EL DESARROLLO DE APLICACIONES WEB A TRAVÉS DE PATRONES DE DISEÑO USANDO EL MARCO DE TRABAJO ZEND FRAMEWORK

DANIEL GUILLERMO CORREA ISAZA
NICOLÁS MESA FERNÁNDEZ

Trabajo de grado para optar al título de

Ingeniero Informático

Santiago Villegas Giraldo
Ingeniero Informático

ESCUELA DE INGENIERÍA DE ANTIOQUIA
INGENIERÍA INFORMÁTICA
ENVIGADO
2012

AGRADECIMIENTOS

Agradecemos al profesor Santiago Villegas Giraldo por las enseñanzas y el acompañamiento en todo el proceso de la elaboración de este trabajo. También queremos agradecerle a nuestro director de carrera, Carlos Jaime Noreña por ayudarnos y aconsejarnos durante la carrera.

CONTENIDO

pág.

LISTA DE FIGURAS

pág.

LISTA DE ANEXOS

pág.

RESUMEN

Para un desarrollador no es fácil saber si está escribiendo código entendible y mantenible en la fase de desarrollo ya que conoce el código entero porque es en lo que ha estado trabajando en los últimos días, semanas o meses. El desarrollador además, no sabe qué tipos de cambios le puedan pedir, entonces lo que termina pasando es que él se da cuenta del nivel de entendibilidad y mantenibilidad de su aplicación mucho después de haberla terminado. Ésto no es ideal.

Por esta razón se necesita encontrar formas de asegurarse de que las aplicaciones desarrolladas si sean mantenibles y entendibles desde la etapa de desarrollo, o incluso desde antes. Para lograrlo, hay que encontrar métricas para evaluar si el código fuente sigue el estándar de codificación de Zend y para ver si es mantenible.

Finalmente se requiere saber cuáles son los patrones de diseño que más mantenibilidad le dan a las aplicaciones, teniendo en cuenta que tienen que estar apegados al estándar de Zend.

En este documento se analizan algunas aplicaciones para buscar problemas en su desarrollo. Luego se documentan varios patrones de diseño que ayudan a generar código mantenible y escalable. Después de ésto, se documentan los estándares de codificación que ayudan a generar código entendible. Finalmente, se analiza la entendibilidad de una aplicación escrita con todos los estándares de codificación encontrados anteriormente.

Palabras clave: Zend framework, patrones de diseño, mantenibilidad, estándares de codificación, entendibilidad

ABSTRACT

It is very difficult for a software developer to know if he/she is writing understandable and maintainable code in the development phase. This is because the developer knows the entire code since he/she has been working on in the last days, weeks or months. The developer doesn't know what type of changes he might have to make, so what usually happens is that he realizes that his code wasn't understandable or maintainable after a long period of time. This is not good.

For this reason, it is important to find ways of making sure that the applications developed are maintainable and understandable since the development phase or even before. To achieve this, it is necessary to find measures to evaluate if the source code is compliant with the Zend codifications standard and to see if it is maintainable.

Finally it is required to know which design patterns provide the most maintainability to applications keeping in mind that they have to be compliant with the Zend coding standard.

This paper documents several design patterns that help programmers to generate maintainable code. Then coding standards from Zend Framework that help to write understandable code are documented. Finally the understandability of an application written with all of the coding standards encountered before is evaluated.

Key words: Zend framework, design patterns, maintainability, coding standards, understandability

INTRODUCCIÓN

Se presenta a continuación algunas aplicaciones en las que se encuentran algunos problemas que pueden ser resueltos con patrones de diseño.

Luego se explica qué es un patrón de diseño y se documentan algunos que ayudan a generar código mantenible.

Después de esto se documentan los estándares de codificación del marco de trabajo de Zend (Zend Framework) que ayudan a generar código entendible.

Finalmente se muestran los resultados de dos aplicaciones: una escrita siguiendo los estándares de codificación y otras sin seguir éstos. Luego se le entregan las aplicaciones a algunos programadores diciéndoles que hagan un cambio y se observan las diferencias entre la modificación de los dos códigos fuente. En seguida se toman conclusiones sobre los resultados obtenidos en la prueba.

1. PRELIMINARES

1.1 PLANTEAMIENTO DEL PROBLEMA

El desarrollo de software hace referencia al conjunto de actividades realizadas desde el inicio hasta la concepción y mantenimiento de artefactos de software. La evolución de la tecnología ha permitido la creación de diferentes tipos de metodologías de desarrollo de software, enfocadas a solucionar diversos problemas que han ido apareciendo a través de los tiempos.

Estas metodologías se han perfeccionado y han generado una nueva cultura donde las personas dedican la mayoría de su tiempo a plasmar requisitos en código fuente, olvidando las buenas prácticas que se deben utilizar a la hora de desarrollar aplicaciones con buena arquitectura y calidad. Como consecuencia, se dificulta el mantenimiento y su reestructuración se convierte en una tarea esencial.

Una forma de disminuir la complejidad de las aplicaciones y lograr mantenibilidad a través de reutilización es hacer uso de patrones de diseño: un conjunto de descripciones diseñadas para dar solución a problemas específicos y concurrentes dentro del contexto computacional.

Se requiere documentar diferentes prácticas y patrones de diseño que se deben utilizar para generar un código entendible y mantenible usando el marco de trabajo de Zend.

1.2 OBJETIVOS DEL PROYECTO

1.2.1 Objetivo General

Documentar patrones de diseño y estilos de programación que deben utilizarse en una aplicación web desarrollada con el marco de trabajo Zend Framework para que sea mantenible, escalable y entendible.

1.2.2 Objetivos Específicos

- Identificar diferentes problemas en el diseño de software de una aplicación web.
- Describir los patrones de diseño más destacados para los problemas más recurrentes que ayudan a generar código mantenible y escalable.
- Identificar las normas de codificación para el estilo de programación establecido por Zend.

- Describir y documentar las normas de codificación que ayudan a generar código entendible.

- Evaluar el nivel de entendibilidad del código fuente a través de una evaluación por personal experimentado sobre una aplicación desarrollada para la prueba.

1.3 MARCO DE REFERENCIA

Todo programador ha sido víctima de que al tener que mantener una aplicación desarrollada por él, no es capaz de entender su propio código. También ocurre mucho que necesita hacer un pequeño cambio, y para hacerlo, le toca cambiar media aplicación.

Desde ese momento es cuando un programador entiende la importancia de desarrollar código mantenible, para que cualquier programador sea capaz de entenderlo y mantenerlo. Desde que surgió la programación orientada a objetos, se encontró una solución a este problema, pero se creó otro problema, la dificultad de este paradigma de la programación.

"Diseñar software orientado a objetos es difícil, y diseñar software reusable orientado a objetos es aún más difícil." (Gamma, Helm, Johnson, & Vlissides, 1994). La diferencia entre los diseñadores experimentados y aquellos que no comprenden el conjunto de problemas que enfrenta el diseño de software es que los diseñadores experimentados tratan de "no resolver cada problema desde el inicio, al contrario, vuelven a utilizar funciones que para ellos han funcionado en el pasado" (Gamma, Helm, Johnson, & Vlissides, 1994). Cuando encuentran una solución, la utilizan una y otra vez, hasta que la solución se convierte en un patrón recurrente que ayuda a construir software mantenible.
Christopher Alexander dijo: "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de la solución para dicho problema, de tal forma que se pueda usar la solución un millón de veces, sin tener que realizarlo de la misma forma dos veces". (Alexander, Ishikawa, Murray, Jacobson, Fiksdahl-King, & Angel, 1977)

El mismo concepto aplica para los patrones de diseño, descripciones de soluciones a problemas, los que se componen de: nombre, problema, solución y consecuencias; Estos componentes ayudan a los programadores a reutilizar soluciones y por ende, mejorar la mantenibilidad de la aplicación.

Otro problema con el que se encuentran los programadores casi que todos los días es con el concepto del *refactoring*. *Refactoring* es básicamente un cambio en el código que intenta mejorar alguna característica del mismo sin modificar su funcionalidad" (Técnicas avanzadas de programación). Todos los programadores tratan de mejorar el código de ellos mismos o de otra persona ya sea para volverlo más eficiente, para corregir un error o para darle una funcionalidad más específica o más general. Cualquiera que sea el caso, es completamente necesario que el código sea entendible para que se sepa qué se está

haciendo y para que sea fácil para el programador darse cuenta si el *refactoring* que le hizo al código si fue bueno.

Otro aspecto que tiene un impacto grande sobre la entendibilidad del código es la documentación de éste. Muchos programadores creen que se van a acordar de qué fue lo que hicieron dos años después y creen que la respuesta se ve obvia en el código. Después de dos años cuando toca hacer un cambio, se llevan la sorpresa de que no entienden lo que habían hecho. Para esto precisamente se crearon los comentarios y la documentación. Al tener dichos comentarios en el código, no solo el creador sino cualquier programador va a poder entender lo que se quería hacer, e incluso se le va a facilitar el tema de *refactoring* si necesita hacerle algún cambio.

2. METODOLOGÍA

A continuación podrá encontrar el procedimiento a seguir para el cumplimiento de los objetivos específicos:

Objetivo 1: Para cumplir con este objetivo, se estudiaron diferentes aplicaciones Web hechas en el marco de trabajo de Zend, y se buscaron los problemas de diseño que tenga en cuanto a mantenibilidad y escalabilidad. Después de analizar varias aplicaciones, se documentaron éstos problemas.

Objetivo 2: Se investigaron los patrones de diseño más conocidos y se estudió cada uno en detalle. Luego se clasificaron para saber cuáles ayudan a los programadores a generar código mantenible y escalable. Los patrones de diseño que cumplían con las dos últimas características serían documentados.

Objetivo 3: Se tomó como referencia el capítulo sobre estándares de codificación de la guía de desarrollo en PHP elaborada para Zend Framework (Framework). En esta guía encontramos indicadores claros de cuáles son los pasos a seguir a la hora de seguir un estándar de codificación para PHP, tales como: demarcación de código, manejo de cadenas de texto, manejo de arrays, declaración de clases, declaración de funciones y métodos, flujo de control, documentación, entre otros.

Objetivo 4: Con base en lo encontrado en el procedimiento anterior, se clasificaron las normas que ayudarían a generar código entendible, y las normas que clasificaran, serían documentadas.

Objetivo 5: Se elaboró una aplicación pequeña siguiendo todo lo encontrado en los procedimientos anteriores. Luego se le entregó el código a varios programadores para que lo leyeran y lo trataran de entender. Se les hizo una encuesta de entendimiento del código para evaluar el nivel de entendibilidad sobre la aplicación.

3. PROBLEMAS EN APLICACIONES DESARROLLADAS EN ZEND FRAMEWORK

3.1 ASTRUM FUTURA

Astrum Futura es un juego de estrategia multijugador escrito en PHP5 usando el framework de desarrollo Zend. La interfaz de usuario usa Javascript y Ajax para la interacción dinámica.

3.1.1 Problemas de diseño

Uno de los principales problemas que se presenta en este desarrollo es que la validación no sigue una línea base que obligue al desarrollador a seguir la buena práctica de validar cada dato de entrada, en vez de esto, el desarrollador manualmente debe agregar los métodos correspondientes de validación.

La desventaja que presenta este diseño a la hora de realizar labores de mantenimiento o en el momento en el que se deseen hacer grandes cambios en que la escalabilidad sea un requisito principal, es que si se presentan muchos cambios en las validaciones, entonces es difícil tener una trazabilidad de cuales campos hay que validar o no.

La evidencia del anterior comportamiento puede observarse a continuación:

```
/*
 * Check request type, and redisplay form if not POST
 */
if($this->getRequest()->getMethod() !== 'POST' || !isset($this->_post))
{
    $this->_forward('signup', 'index');
    return;
}

/*
 * Get filtered input data or else exit current action
 * The associated errorHandler will have setup
 * additional instructions for handling bad input
 */
if(!$clean = $this->validateProcessInput())
{
    return;
}
```

Figura 1: Recepción de parámetros a través del método POST

```php
/**
 * Validate input data passed to processAction method.
 *
 * @access private
 * @return mixed Boolean false if invalid data, or clean input array
 */
private function validateProcessInput()
{
    $clean = array();

    /*
     * Validate all expected input values
     */
    if(!$clean['user'] = $this->_post->testRegex('astrum_form_game_name', self::REGEX_NAME))
    {
        return $this->handleError('astrum_form_game_name');
    }
    if(!$clean['fleet'] = $this->_post->testRegex('astrum_form_game_fleet', self::REGEX_NAME))
    {
        return $this->handleError('astrum_form_game_fleet');
    }
    return $clean;
}
```

Figura 2: Validación manual de los datos de entrada

3.2 DIGITALUS CMS

Digitalus es un sistema de administración de contenido desarrollado en Zend Framework que soporta las siguientes características:

- Planeación de proyecto
- Diseño creativo
- Administración de contenido
- Internacionalización
- Medios de comunicación social
- Comercio electrónico
- Desarrollo de aplicaciones a medida

Según el sitio web, “Al menos 80.000 desarrolladores están usando nuestra versión abierta del CMS” (Digitalus), lo que es un indicador de que el sistema es bien conocido y usado en muchos desarrollos alrededor del mundo.

3.2.1 Problemas de diseño

Todas las sentencias ejecutadas en esta aplicación sobre la base de datos se realizan a través de sentencias estáticas; dichas sentencias están diseñadas solo para ejecutarse en un motor de base de datos MySQL.

Cuando los requisitos de negocios establezcan un cambio de motor de base de datos, el desarrollador debe hacer el cambio de cientos de líneas de código para adaptarse a la nueva necesidad. Claramente la escalabilidad y mantenibilidad de la aplicación es afectada por esta mala práctica de programación.

La evidencia del uso de sentencias estáticas puede ser observada a continuación:

```
/**
 * this function returns  all of the admin access
 *
 * @return zend_db_rowset
 */
public function adminAccess($limit = 50)
{
    $sql = "SELECT
        users.first_name,
        users.last_name,
        users.role,
        traffic_log.page,
        traffic_log.ip,
        FROM_UNIXTIME(traffic_log.timestamp) as date
        FROM
        traffic_log
        Inner Join users ON traffic_log.user_id = users.id
        ORDER BY timestamp DESC
        LIMIT {$limit}";
    return $this->_db->fetchAll($sql);
}
```

Figura 3: Sentencias estáticas para la ejecución en MySQL

El problema para este caso, es el uso de la instrucción LIMIT, el cual no es contemplado en el estándar SQL y por ende, su implementación no es obligatoria en todos los motores de bases de datos que siguen este estándar. Uno de estos motores es Oracle, el cual no hace uso de la instrucción LIMIT, sino de la palabra reservada rownum.

4. PATRONES DE DISEÑO

4.1 DEFINICIÓN

Según Christopher Alexander, un patrón “describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de la solución a ese problema, de tal forma que usted pueda usar esa solución un millón de veces” (Alexander, Ishikawa, Murray, Jacobson, Fiksdahl-King, & Angel, 1977).

“A pesar de que Alexander hablaba acerca de patrones en edificios y pueblos lo que él dice también aplica a los patrones de diseño orientados a objetos” (Gamma, Helm, Johnson, & Vlissides, 1995).

4.2 ELEMENTOS

En 1995 un grupo de cuatro programadores conocidos actualmente como la “Pandilla de los cuatro”, notaron que había muchos obstáculos en el desarrollo de software que se presentaban muy seguido. Por esta razón, esta pandilla decidió buscar la mejor solución para pasar estos obstáculos, de forma que se pudiera utilizar cada vez que se encontrara el obstáculo. En ese momento produjeron un libro “Design Patterns: Elements of Reusable Object Oriented Sofware” que lista 23 de estas soluciones llamadas patrones de diseño. (DaveOrme, Marzo)

La pandilla de los cuatro establece cuatro elementos esenciales que debe tener un patrón de diseño. (Gamma, Helm, Johnson, & Vlissides, 1995).

1. **Nombre:** Se usa para describir el problema de diseño, las soluciones y consecuencias en una o dos palabras.

2. **Problema:** Describe las condiciones para la aplicabilidad del patrón y la lista de condiciones que deben cumplirse antes de que el patrón tenga sentido.

3. **Solución:** Describe los elementos, relaciones, responsabilidades, y colaboraciones que hacen posible la solución del problema de una forma abstracta. La solución no debe describir un diseño o implementación en particular.

4. **Consecuencias:** Son los resultados y mejoras de la aplicación del patrón.

5. PATRONES DE DISEÑO DESTACADOS

5.1 ADAPTER

5.1.1 Objetivo

“Convertir la interfaz de una clase en otra interfaz que el cliente espera. Le permite a las clases trabajar conjuntamente cuando de otro modo no serían compatibles” (Gamma, Helm, Johnson, & Vlissides, 1995).

5.1.2 Problema

Existen componentes que ofrecen funcionalidades que pueden ser utilizadas en la aplicación, pero su incompatibilidad con la arquitectura del sistema actual hace necesaria la creación de una interfaz que permita adaptar el componente a la necesidad requerida.

Otro problema que puede ser solucionado a través del adaptador es cuando se desea crear una clase reusable que trabaje conjuntamente con clases que no tienen interfaces compatibles.

5.1.3 Ejemplo

A pesar de la existencia de un estándar para consultas (SQL), la gran cantidad de motores de bases de datos y sus propias implementaciones hacen que el programador esté obligado a desarrollar aplicaciones para un motor específico. Un cambio de motor implicaría cambios en cientos de líneas de código lo cual hace que la aplicación sea poco mantenible.

Usando la solución que provee el adaptador, se puede construir una interfaz que permita, por medio de métodos comunes, acceder de forma genérica a cualquier función del motor de base de datos.

La figura 1 muestra la representación de la solución usando diagrama de clases. La clase abstracta *DbAdapter* define los métodos comunes que deben ser implementados para cada motor de base de datos. En el ejemplo son implementados los métodos básicos que son: creación, lectura, actualización y borrado.

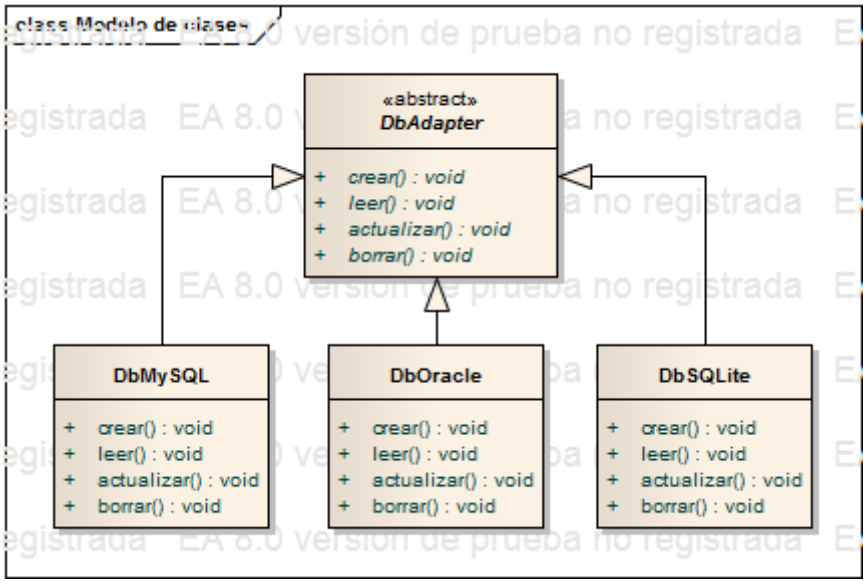


Figura 4: Modelo genérico de patrón Adapter

5.1.4 Código de ejemplo

La recolección de metadatos de una base de datos es un ejemplo perfecto para probar la utilidad del patrón adapter. En cada motor, la manera de consultar metadatos es diferente, lo que hace que un cambio de motor obligue al desarrollador a realizar cambios grandes, afectando la mantenibilidad.

Para este ejemplo se utilizan dos motores de base de datos: MySQL y SQLite. Se pretende extraer los metadatos de dos bases de datos (una en cada motor) y la información extraída mostrarla en una tabla al usuario.

A continuación se muestra el código utilizado para esta demostración:

Configuración

```
resources.db.adapter = "pdo_mysql"
resources.db.params.host = "localhost"
resources.db.params.username = "example"
resources.db.params.password = "example"
resources.db.params.dbname = "adapter"

;Para extraer los datos de SQLite, se quitan los comentarios
;en las siguientes líneas
;resources.db.adapter = "pdo_sqlite"
;resources.db.params.dbname = APPLICATION_PATH
"/../database/adapter.db"
```

Controlador

```
/* Se coge el adaptador de base de datos */
$db = Zend_Db_Table::getDefaultAdapter();

$results = array();

/*
 * Se llama la función listTables para listar las tablas
 * y meterlas en un vector
 */
 $tables = $db->listTables();

/*
 * Se itera por las tablas guardando en otro vector los  +
 * metadatos de cada tabla
 */
 foreach ($tables as $table){
    $results[$table] = $db->describeTable($table);
 }

 /* Se le mandan los resultados a la vista */
 $this->view->results = $results;
```

Este controlador se puede usar para cualquiera de los dos motores de base de datos sin tener que cambiar una sola línea de código, lo cual demuestra la utilidad de este patrón.

5.1.5 Resultados

MySQL

Tablas	Columnas
blog	id(int), name(text), title(text)
news	id(int), title(text), description(text)
users	id(int), username(varchar), password(varchar)

Figura 5: Resultado de la extracción de metadatos de una base de datos en un motor MySQL

Las consultas realizadas para obtener este resultado fueron:

```
SHOW TABLES;
DESCRIBE `blog`;
DESCRIBE `news`;
DESCRIBE `users`;
```

SQLite

Tablas	Columnas
Blog	id(INTEGER), name(TEXT), title(TEXT)
News	id(INTEGER), title(TEXT), description(TEXT)
Users	id(INTEGER), username(TEXT), password(TEXT)

Figura 6: Resultado de la extracción de metadatos de una base de datos en un motor SQLite

Las consultas realizadas para obtener este resultado fueron:

```
SELECT name FROM sqlite_master WHERE type='table' UNION ALL SELECT
name FROM sqlite_temp_master WHERE type='table' ORDER BY name;
PRAGMA table_info("Blog");
PRAGMA table_info("News");
PRAGMA table_info("Users");
```

5.1.6 Análisis de resultados

Mirando las consultas realizadas, se puede ver que son completamente distintas pero de todas formas, el resultado muestra que se pudo mostrar sin ningún problema usando el mismo código. Lo único que hubo que cambiar fue el archivo application.ini en el que se puso la nueva configuración de la base de datos.

Este mismo caso aplica para cualquier consulta que se haga en diferentes motores de base de datos. Si este patrón se usa adecuadamente, se podría ahorrar varias horas de trabajo y sobrecostos a la hora de cambiar de motor de base de datos. Además de ésto, si no hay que hacer cambios en toda la aplicación, ésta estará menos propensa a tener nuevos errores.

5.2 COMPOSITE

5.2.1 Objetivo

“Componer objetos en estructuras de árbol para representar jerarquías de una parte o un todo. Le permite al cliente tratar uniformemente objetos individuales y composiciones de objetos” (Gamma, Helm, Johnson, & Vlissides, 1995)

5.2.2 Problema

En general se desean resolver dos problemas:

1. Se requiere representar una parte o un todo de una jerarquía.
2. Se requiere ignorar la diferencia entre composiciones de objetos y objetos individuales.

5.2.3 Ejemplo

Un ejemplo práctico para la demostración de la utilidad del patrón es el uso de formularios orientados a objetos. A través del patrón podemos realizar fácilmente tareas sobre formularios que por el contrario serían difíciles de implementar.

Algunas de estas tareas son:

1. Filtrar y validar entradas de usuario.
2. Ordenar elementos en el formulario.
3. Presentación en la vista de los elementos.
4. Agrupación de formularios.
5. Configuración del nivel de los elementos y los formularios.

5.2.4 Código de ejemplo

El uso del patrón composite se da naturalmente al usar las clases para formulario y elementos que son nativas del marco de desarrollo.

En el ejemplo se creará un formulario de autenticación, que tendrá como entradas el correo electrónico y la contraseña de la persona. Diferentes validaciones y filtros se realizarán al tratar uniformemente el formulario.

Formulario de autenticación

```
<?php

/*
 * El formulario Form_Authentication contiene los elementos
 * que se tratarán uniformemente
 */
class Form_Authentication extends Zend_Form {
    public function init() {

        /* El elemento email tendrá un validador y un filtro:
         * - Validador de formato de correo electrónico
         * - Filtro para eliminar espacios al inicio y
         * final del correo
         */
        $email = new Zend_Form_Element_Text('email');
        $email->setRequired(true)
              ->addValidator(new Zend_Validate_EmailAddress())
              ->addFilter(new Zend_Filter_StringTrim());

        /* El elemento de contraseña solo tiene un validador:
         * - Se valida que la contraseña tena 8 o más caracteres
         * - Al menos una letra minúscula, mayúscula,
         * dígito y especial
         */
        $regex = '/(?=^.{8,}$)(?=.*\d)(?=.*\W+)(?=.*[A-Z])(?=.*[a-z]).*$/';
        $password = new Zend_Form_Element_Password('password');
        $password->setRequired('true')
                 ->addValidator(new Zend_Validate_Regex($regex));

        // El element submit, el cual envia los datos al backend
        $submit = new Zend_Form_Element_Submit('submit');


        $elements = array(
            $email,
            $password,
            $submit
        );

        /* Se agregan todos los elementos al formulario */
        $this->addElements($elements);
    }
}
```

Controlador

El controlador se encarga de crear el formulario, asignarlo a la vista y procesar todos los datos de entrada.

```
/* Obtiene los datos enviados por el usuario */
$request = $this->_request;

/* Se crea el nuevo formulario */
$form = new Form_Authentication();

/* Si el usuario envío datos */
if ($request->isPost()) {

    /* Entonces, el formulario se encarga de validar
     * todos los datos:
     * - Aplica todos los filtros de cada elemento
     * - Aplica todos los validadores de cada elemento
     */
    if ($form->isValid($request->getPost())) {

    } else {
        /* Si algún datos falla la validación
         * entonces se repobla el formulario con:
         * - El contenido de las entradas
         * - El error para cada elemento si lo hay
         */
        $form->populate($request->getPost());
        $this->view->form = $form;
    }
} else {
    $this->view->form = $form;
}
```

5.2.5 Resultados

Formulario sin envío de datos

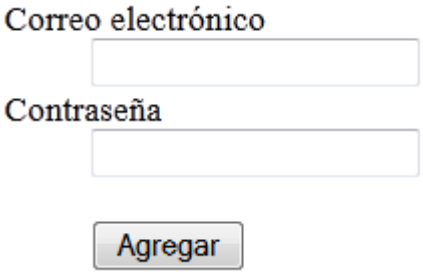


Figura 7: Formulario sin envío de datos

Formulario con envío de datos

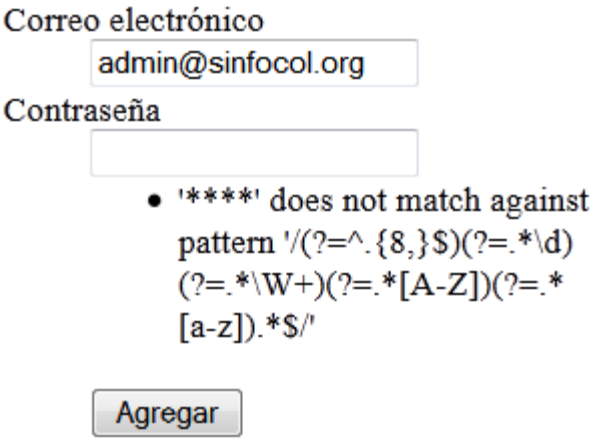


Figura 8: Formulario con envío de datos donde uno de ellos falla la validación

5.2.6 Análisis de resultados

El patrón composite es muy útil a la hora de usar un formulario, ya que permite tratar el formulario y todos sus elementos uniformemente. En el momento en el que el método isValid del formulario es llamado, éste itera sobre todos sus hijos (Elementos o composiciones de elementos), y llama el método isValid de cada uno de ellos, la automatización de validaciones, filtros y cualquier otro método posible es sencilla.

5.3 DECORATOR

5.3.1 Objetivo

“Agregar responsabilidades adicionales a un objeto de manera dinámica. Es una alternativa a la herencia para extender la funcionalidad de una clase” (Gamma, Helm, Johnson, & Vlissides, 1995).

5.3.2 Problema

Un programador quiere cambiar el comportamiento para algunos objetos de una clase pero no para todos. La herencia es una solución pero trae con ella el problema de ser estática. Si es necesario agregar otros comportamientos se tendría varias herencias (una por cada comportamiento). Además, si se necesita que algunos objetos tengan dos de estos comportamientos comienzan a surgir problemas.

Es mejor dejar la clase original y hacer una envoltura que se encargue de agregarle al objeto el comportamiento deseado. De esta forma, se puede agregar varias envolturas a un mismo objeto agregándole todos los comportamientos que se desee en tiempo de ejecución.

5.3.3 Ejemplo

Se requiere mostrar un formulario en diferentes partes, y en cada una de estas partes, mostrarlo de formas distintas.

5.3.4 Código de ejemplo

El patrón decorator en los formularios de Zend Framework es ideal para que el programador decida como presentar dicho formulario. Para hacer esto, "se usa un patrón decorator modificado; en lugar de decorar el objeto, se decora el contenido generado. El patrón decorator opera desde afuera hacia adentro mientras éste opera desde adentro hacia afuera." (Zend Developer Zone, 2008)

El ejemplo creará un solo formulario de autenticación el cual se decorará de diversas formas para mostrar la utilidad de este patrón.

Formulario de autenticación

```
<?php
class Form_Login extends Zend_Form {
    private $decorator;

    /**
     * 'table' se presenta como table, 'list' se presenta
```

```
     * como lista
     * @param string $renderType
     */
    public function __construct($renderType = null) {
        /* Se verifica el decorador a agregar al formulario */
        switch ($renderType) {
            case 'table':
                $this->decorator = 'login-table.phtml';
                break;
            case 'list':
                $this->decorator = 'login-list.phtml';
                break;
            default:
                break;
        }
        parent::__construct();
    }

    public function init() {
        $this->setMethod('post');

        $user = new Zend_Form_Element_Text('username');
        $user->setRequired(true)
                ->setLabel('Nombre de usuario');

        $pass = new Zend_Form_Element_Text('password');
        $pass->setRequired(true)
                ->setLabel('Contraseña');

        $submit = new Zend_Form_Element_Submit('submit');
        $submit->setIgnore(true)
                ->setLabel('Agregar');

        $elements = array($user, $pass, $submit);

        $this->addElements($elements);

        /* Se verifica si se escogió un decorador */
        if ($this->decorator) {
            /* Se eliminan los decoradores por defecto */
            foreach ($this->getElements() as $element) {
                $element->removeDecorator('Errors');
                $element->removeDecorator('DtDdWrapper');
                $element->removeDecorator('Label');
                $element->removeDecorator('HtmlTag');
            }
```

```
            /* Se pone el decorador escogido */
            $this->setDecorators(array(
                array('ViewScript',
                array('viewScript' => 'templates/' . $this-
>decorator))
            ));
        }
    }
}
```

Decorador para mostrar el formulario en una tabla

En este decorador se coge el formulario y pone cada campo en una tabla.

```
<?php $form = $this->element; ?>

<form action="<?php echo $this->escape($form->getAction()); ?>"
method="<?php echo $this->escape($form->getMethod()) ?>">
    <table border="1">
        <tr>
            <td>Usuario</td>
            <td><?php echo $form->username ?></td>
        </tr>
        <tr>
            <td>Contraseña</td>
            <td><?php echo $form->password ?></td>
        </tr>
        <tr>
            <td colspan="2"><?php echo $form->submit ?></td>
        </tr>
    </table>
</form>
```

Decorador para mostrar el formulario en una lista

En este decorador se coge el formulario y pone cada campo en una lista.

```
<?php $form = $this->element; ?>

<form action="<?php echo $this->escape($form->getAction()); ?>"
method="<?php echo $this->escape($form->getMethod()) ?>">
    <ul>
        <li>Usuario <?php echo $form->username ?></li>
        <li>Contraseña <?php echo $form->password ?> </li>
        <li><?php echo $form->submit ?></li>
    </ul>
```

```
</form>
```

Controlador

En el controlador se decide el decorador por usar. Luego se instancia el formulario de autenticación pasando la decisión anterior como parámetro. Luego se le pasa el formulario a la vista para mostrárselo al usuario.

```
/* Se saca parámetro type de la url para saber el tipo de
 * decorador a utilizar en el formulario
 */
$type = $this-> request->getParam('type');

/* Se instancia el formulario pasando el tipo como parámetro */
$form = new Form_Login($type);

/* Se le pasa el formulario a la vista */
$this->view->form = $form;
```

Vista

En la vista solo hay que imprimir el formulario en pantalla sin preocupaciones por cómo está compuesto el formulario por dentro.

```
<?php
    echo $this->form;
```

5.3.5 Resultados

Formulario sin decorador

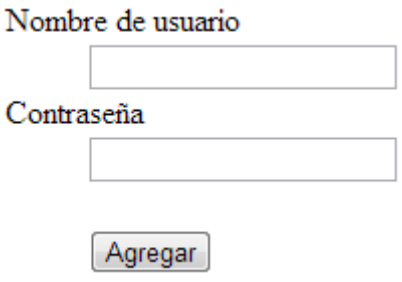


Figura 9: Formulario sin decorador

Formulario con decorador de tabla

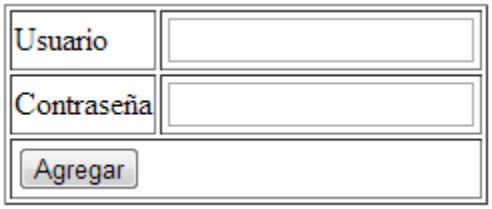


Figura 10: Formulario con decorador de tabla

Formulario con decorador de lista

Figura 11: Formulario con decorador de lista

5.3.6 Análisis de resultados

El patrón decorator es muy útil para decorar un formulario sin cambiar su estructura, codificación o tener que heredar. Como se puede ver en el controlador, lo único que se hace es escoger el tipo de decorador a utilizar y mostrar el formulario en pantalla. El decorador se encarga de mostrar el formulario tal como fue configurado haciendo posible la reutilización del mismo formulario en diferentes tipos de vistas.

5.4 REGISTRY

5.4.1 Objetivo

“Disponer de un objeto bien conocido el cual otros pueden usar para encontrar objetos comunes y servicios”. (Fowler, Rice, Foemmel, Hieatt, Mee, & Stafford, 2002)

5.4.2 Problema

La mayoría de movimientos entre objetos es realizada al seguir referencias de sus campos. Si se desea encontrar todas las órdenes para un cliente, la mayoría de veces se debe empezar con el objeto cliente y usar sus métodos para obtener las órdenes.

En cualquier caso, hay ocasiones donde no se tiene un objeto apropiado para comenzar, por lo cual se necesita usar algún tipo de método de búsqueda, para lo cual el patrón registry es útil.

Registry es esencialmente un objeto global que permite la búsqueda de objetos y servicios comunes.

5.4.3 Ejemplo

Uno de los principales problemas en las aplicaciones web es la falta de registro para eventos excepcionales. El problema es resuelto al usar un objeto global que permita registrar el tiempo exacto, el nivel de severidad y mensajes descriptivos de los eventos generados por la aplicación.

5.4.4 Código de ejemplo

Para el uso correcto del patrón Registry en el marco de desarrollo Zend, usamos la clase Zend_Registry, la cual implementa la funcionalidad descrita en la sección problema.

En el archivo bootstrap inicializamos la clase Zend_Log y agregamos la clase Zend_Log, el cual será el objeto común que puede ser usado en cualquier parte de la aplicación.

Bootstrap.php

```
protected function _initLogger()
{
    /* Inicializamos la clase Zend_Log, la bitácora donde se
     * registrarán los errores es error_log
     */
    $writer = new Zend_Log_Writer_Stream('error_log');
    $logger = new Zend_Log($writer);
```

```
    /* Agregamos el objeto común en el Registry */
    Zend_Registry::set('logger', $logger);
}
```

Controlador

Desde el controlador podemos utilizar de una forma fácil el objeto logger:

```
public function registryAction()
{
    /* Obtenemos el objeto logger de la clase Registry */
    $logger = Zend_Registry::get('logger');
    /* Almacenamos el error en la bitácora */
    $logger->log('Bruteforce detected', Zend_Log::ALERT);
}
```

5.4.5 Resultados

En el caso de ocurrencia de un evento excepcional, es posible usar fácilmente el objeto logger, ya que este se encuentra como un objeto común disponible para toda la aplicación.

Al detectar un evento el registro correspondiente es almacenado en el archivo error_log, cuyo contenido como ejemplo es mostrado a continuación:

```
2012-10-10T14:38:30-05:00 ALERT (1): Bruteforce detected
2012-10-10T14:40:35-05:00 ALERT (1): Bruteforce detected
2012-10-10T14:46:19-05:00 ALERT (1): Bruteforce detected
```

Figura 12: Evento excepcional registrado usando Registry

5.4.6 Análisis de resultados

El patrón Registry es una forma poderosa de tener un punto de partida para la búsqueda de objetos cuya visibilidad debe ser para toda la aplicación. Vemos como el acceso al objeto para registro de eventos se hace de una forma inmediata.

Como todo patrón de diseño, debe buscar el momento adecuado para implementarlo, los objetos visibles para toda la aplicación pueden ser abusados. Martin Fowler, creador del patrón, recomienda solo usar el patrón Registry cuando este sea necesario, o sea, en el momento de requerir objetos comunes globales y uso de servicios.

5.5 FRONT CONTROLLER

5.5.1 Objetivo

“Crear un controlador que permita manejar todas las peticiones de un sitio web.” (Fowler, Rice, Foemmel, Hieatt, Mee, & Stafford, 2002)

5.5.2 Problema

Cuando no hay un punto de acceso centralizado en la aplicación, los esfuerzos de cambio son mayores, afectando la mantenibilidad. “En un sitio web completo, hay muchas cosas que usted debe hacer en el momento de manejar una petición. Entre ellas podemos incluir el manejo de la seguridad, la internacionalización, proveer de una vista particular para ciertos tipos de usuarios, entre otros” (Fowler, Rice, Foemmel, Hieatt, Mee, & Stafford, 2002)

Si el controlador de entrada se encuentra a través de múltiples objetos, entonces muchos de los comportamientos pueden ser duplicados, y finalmente es difícil cambiar dicho comportamiento en tiempo de ejecución.

5.5.3 Ejemplo

Un sitio en funcionamiento a nivel mundial requiere la traducción de las rutas en la URL para extender su mercado a todos los públicos posibles. Un requisito de este estilo normalmente demanda recursos exagerados de tiempo y dinero cuando no se tiene una arquitectura donde las peticiones están centralizadas.

El patrón de Front Controller da respuesta al requisito al centralizar todas las peticiones a través de un solo objeto.

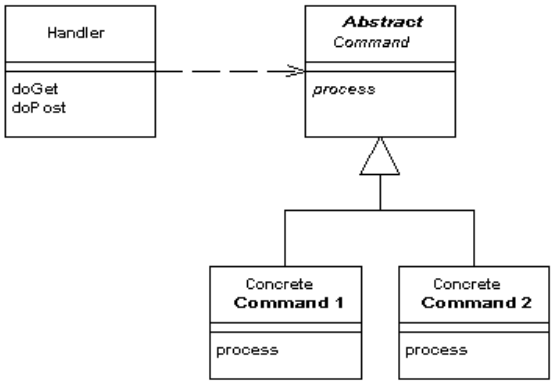


Figura 13: Patrón Front Controller

Este objeto será el encargado de traducir todas las rutas sin la necesidad de crear un archivo por lenguaje.

5.5.4 Código de ejemplo

En el archivo bootstrap.php obtenemos el objeto del Front Controller instanciado por los mecanismos internos de Zend, y sobreescribimos las rutas manejadas por dicho controlador para habilitar el soporte de rutas en diferentes idiomas.

bootstrap.php

```
protected function _initInternationalization()
{
    // Se obtiene el Front Controller de la aplicación
    $front  = Zend_Controller_Front::getInstance();
    // Obtenemos el enrutador por defecto
    $router = $front->getRouter();

    // Creamos dos nuevos traductores para las rutas: español y
alemán
    $translator = new Zend_Translate('array', array(''));
    $translator->addTranslation(
        APPLICATION_PATH . '/languages/route_es.php', 'es');
    $translator->addTranslation(
        APPLICATION_PATH . '/languages/route_de.php', 'de');

    /* Nota: En este punto se debe detector el lenguaje deseado
     *       para el usuario, puede ser a través de cabeceras
     *       del navegador, o configuraciones predeterminadas.
     * Para el ejemplo por defecto se usa la traducción 'de'
     */
    $translator->setLocale('de');

    // Creamos una ruta para el controlador
    $controller = new
Zend_Controller_Router_Route(':@controller/*');
    $controller->setDefaultTranslator($translator);

    // Creamos una ruta para el controlador y la acción
    $controller_action =
        new
Zend_Controller_Router_Route(':@controller/:@action/*');
    $controller_action->setDefaultTranslator($translator);

    // Agregamos las rutas al enrutador por defecto
    // Front Controller se encarga de manejar las rutas
    $router->addRoute('controller', $controller);
```

```
    $router->addRoute('controller_action', $controller_action);
}
```

route_es.php

```
<?php

// El índice del array es el nombre del controlador o acción real
usado
// El valor es la traducción
return array(
    'archive' => 'archivo',
    'user'    => 'usuario',
    'about'   => 'acerca_de',
    'index'   => 'indice'
);
```

route_de.php

```
<?php

// El índice del array es el nombre del controlador o acción real
usado
// El valor es la traducción
return array(
    'archive' => 'archiv',
    'user'    => 'benutzer',
    'about'   => 'uber',
    'index'   => 'uebersicht',
);
```

views/script/user/index.phtml

```
Testing translated route for user controller
```

5.5.5 Resultados

Cuando el lenguaje usado es alemán, podemos ingresar al controlador user, acción index, a través de la ruta /benutzer/uebersicht/, como lo indica la siguiente figura:

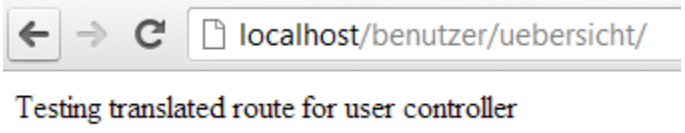


Figura 14: Acceso al controlador user a través de ruta en alemán

Cuando el lenguaje usado es español, podemos ingresar al controlador user, acción index, a través de la ruta /usuario/índice/, como lo indica la siguiente figura:

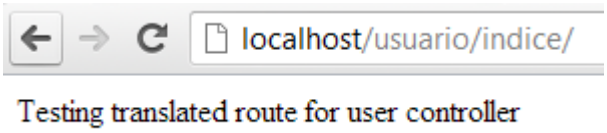


Figura 15: Acceso al controlador user a través de ruta en español

5.5.6 Análisis de resultados

El patrón Front Controller ofrece la principal ventaja de poder tener a mano todas las peticiones que ocurren en la aplicación web, esta ventaja permite implementar requisitos funcionales y no funcionales que normalmente tiene una demanda alta de recurso en tiempo y dinero.

A través de rutas traducidas podemos ofrecerle al usuario un mejor entendimiento de la aplicación, sin tener que agregar controladores y acciones extras que dupliquen funcionalidad en la aplicación. Este patrón aumenta la escalabilidad de cualquier aplicación al tener el punto central del manejo de peticiones.

5.6 FACTORY METHOD

5.6.1 Objetivo

"Definir una interfaz para crear un objeto, pero dejar que las subclases decidan cual objeto instanciar. Factory Method permite entregarle la responsabilidad de instanciar a las subclases. " (Gamma, Helm, Johnson, & Vlissides, 1995)

5.6.2 Problema

"Los frameworks usan clases abstractas para definir y mantener relaciones entre objetos. Un framework a menudo es responsable de crear estos objetos también." (Gamma, Helm, Johnson, & Vlissides, 1995)

El problema es que el marco de desarrollo solo sabe que tiene que crear un objeto nuevo pero no sabe qué tipo de objeto. "Esto crea un dilema: El framework tiene que instanciar clases, pero solo sabe de clases abstractas, que no pueden ser instanciadas." (Gamma, Helm, Johnson, & Vlissides, 1995)

La solución que propone este patrón es encapsular la lógica de cual clase crear y retornar el objeto listo para ser utilizado.

5.6.3 Ejemplo

En las aplicaciones web, hacer caché es vital para mejorar el rendimiento y evitar correr funciones muy pesadas repetidas veces. Hay tres modos distintos de hacer caché:

- Caché a toda una página (page): Básicamente guarda la página entera y cada que se hace una petición por esa página se retorna la de la caché evitando que el servidor la vuelva a procesar.
- Caché a las salidas echo (output): Se le hace caché a algunas porciones de la página.
- Caché al retorno de funciones (core): Le hace caché a lo que va a retornar una función para que cuando se vuelva a llamar, no se tenga que volver a procesar toda la función.

El problema es que el framework no tiene forma de saber el tipo de caché a instanciar. Zend Framework soluciona este problema haciendo uso del patrón Factory Method. De esta manera, se le pone en un parámetro el tipo de caché que se desea utilizar y éste retorna un objeto caché con las especificaciones requeridas.

5.6.4 Código de ejemplo

Controlador

```
public function factoryAction() {

    /* Se especifica la duración de la caché antes de que
     * sea necesario volver a ejecutar la función.
     */
    $frontendOptions = array(
        'lifetime' => 30
    );

    // Se especifica la ubicación del directorio para hacer caché
    $backendOptions = array(
        'cache_dir' => APPLICATION_PATH . '/tmp/'
    );

    /* Se mandan parámetros al Factory para que se haga
     * el objeto cache con
     * las especificaciones requeridas
     *
     * Se usa caché de tipo core
     */
    $cache = Zend_Cache::factory('Core', 'File', $frontendOptions,
$backendOptions);

    /* Se guarda el tiempo antes de que empiece a ejecutar el
ciclo */
    $startTime = microtime(true);

    /* Se verifica si ya está en caché antes de ejecutar el ciclo
*/
    if (($data = $cache->load('example')) === false) {
        /* No estaba en caché */
        $data = '';
        for ($i = 0; $i < 10000; $i++) {
            if ($i % 1000 === 0) {
                /* Se duerme un segundo cada 1000 iteraciones
                 * para que la función sea más lenta y se note
                 * la direrencia con el funcionando
                 */
                sleep(1);
                $data = $data . $i . '<br/>';
            }
        }
```

```
        /* Se almacena el tiempo en el momento que se
         * crea la caché
         */
        $data .= '<b>Se crea la caché : </b>' . time() . '<br/>';

        $cache->save($data);
    }

    /* Se agrega el tiempo actual */
    $data .= '<b>Tiempo actual: </b>' . time();

    /* Se calcula y se muestra el tiempo de ejecución */
    $executionTime = microtime(true) - $startTime;
    $data .= '<br/><br/><b>Tiempo de ejecución: </b>' .
             $executionTime;

    /* Se mandan los datos a la vista */
    $this->view->data = $data;
}
```

Se obtiene la instancia de caché, se mira si existe algo en caché con el identificador "example". Si existe se muestra éste; si no, se procesa todo el ciclo, y se guarda en caché cuando se termina.

Vista

```
<?php echo $this->data; ?>
```

Se muestra en pantalla los datos que manda el controlador

5.6.5 Resultados

Toda la lógica para saber qué tipo de objeto instanciar y con cuales características, la hizo el patrón Factory Method. Después de crear el objeto, el programador tiene acceso a todos sus métodos para poder hacer caché a lo que necesite. En este caso se usó para hacer caché de un ciclo largo y lento.

A continuación se muestran dos resultados: antes de que el resultado del ciclo quede en caché, y después de que quede en caché:

0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Se crea la caché : 1350326932
Tiempo actual: 1350326932

Tiempo de ejecución: 10.016840934753

Figura 16: Sitio cargado antes de hacer caché

0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Se crea la caché : 1350326932
Tiempo actual: 1350326946

Tiempo de ejecución: 0.0016779899597168

Figura 17: Sitio cargado después de hacer caché

5.6.6 Análisis de resultados

En el código del controlador, se puede ver que no hay lógica de qué tipo de caché instanciar, pues el Factory se encarga de hacer esta lógica y entregarle el objeto con las especificaciones que se requieren.

En el ejemplo también se puede ver la utilidad de Zend_Cache que bajó un método de más de 10 segundos en ejecución a uno de menos de un segundo, lo que hace que la aplicación sea más rápida y ayuda a evitar que el servidor se sature por procesamiento.

5.7 TABLE DATA GATEWAY

5.7.1 Objetivo

“Disponer de un objeto que actúa como puerta de enlace a una tabla en la base de datos.” (Fowler, Rice, Foemmel, Hieatt, Mee, & Stafford, 2002)

5.7.2 Problema

“Mezclar SQL en la lógica de la aplicación puede causar severos problemas. Muchos desarrolladores no se sienten cómodos con el SQL, y muchos otros que están cómodos pueden no escribirlo bien.” (Fowler, Table Data Gateway)

El patrón de Table Data Gateway permite definir una interfaz donde se define operaciones comunes sobre tablas en la base de datos, tales como inserción, lectura, borrado y actualización de datos. Otros métodos usan esta puerta de enlace para la interacción con la base de datos.

Cada método en el Table Data Gateway apunta sus parámetros hacía una petición SQL, la cual ejecuta contra la base de datos, este comportamiento convierte al Table Data Gateway en un patrón sin estado, ya que su único rol es empujar y retornar datos.

5.7.3 Ejemplo

“La clase Zend_Db_Table consiste en una interfaz orientada a objetos de una tabla en la base de datos. Provee métodos comunes para operaciones sobre tablas. La clase base es extendible, se puede agregar lógica personalizable.

Es una implementación del patrón Table Data Gateway. La solución también incluye una clase que implementa el patrón Row Data Gateway”. (Zend)

La representación en modelo de clases para el patrón puede observarse a continuación y se implementará en la sección de código de ejemplo:

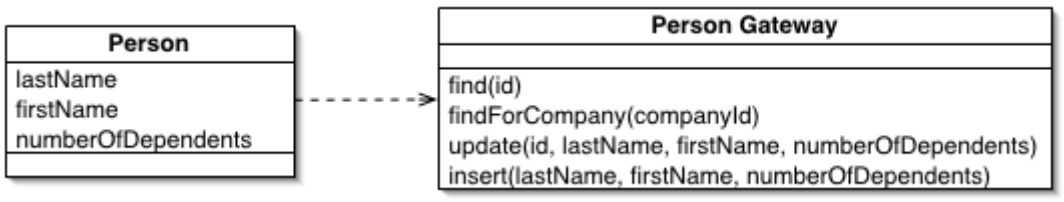


Figura 18: Representación Table Data Gateway

5.7.4 Código de ejemplo

En el application.ini configuramos adecuadamente la conexión a la base de datos:

application.ini

```
resources.db.adapter = "PDO MYSQL"
resources.db.params.dbname = "example"
resources.db.params.username = "test"
resources.db.params.password = "test"
resources.db.params.host = "localhost"
```

El controlador contiene la lógica del programa que implementa el patrón:

Controlador

```
public function gatewayAction() {
    /* Se instancia la nueva puerta de enlace usando
     * la tabla "person"
     */
    $table = new Zend_Db_Table('person');

    // Creamos 10 iteraciones
    for ($i = 0; $i < 10; $i++) {

        // Creamos nuevos valores para la tabla "person"
        $data = array(
                    'firstName' => 'first' . rand(),
                    'lastName'  => 'last'  . rand(),
                    'numberOfDependents' => rand()
                );
        // Insertamos los datos usando el método insert
        $table->insert($data);
    }

    // Establecemos los nuevos valores para el ID 1
    $data = array(
                'firstName' => 'firstModified',
                'lastName'  => 'lastModified',
                'numberOfDependents' => 0
            );
    $where = $table->getAdapter()->quoteInto('id = ?', 1);
    // Actualizamos los datos del ID 1
    $table->update($data, $where);

    $where = $table->getAdapter()->quoteInto('id = ?', 2);
    // Eliminamos los datos del ID 2
```

```
    $table->delete($where);

    $select = $table->select();
    $select->where('id = ?', 5);
    // Seleccionamos todas las personas con id igual a 5
    $rows = $table->fetchAll($select);

    foreach ($rows as $row) {
        echo $row['id'] . ': ' . $row['firstName'] . ' ' .
             $row['lastName'] . PHP_EOL;
    }

    // Buscamos la persona con el ID 10 (Basado en la llave
primaria)
    $rows = $table->find(10);
    foreach ($rows as $row) {
        echo $row['id'] . ': ' . $row['firstName'] . ' ' .
             $row['lastName'] . PHP_EOL;
    }
}
```

5.7.5 Resultados

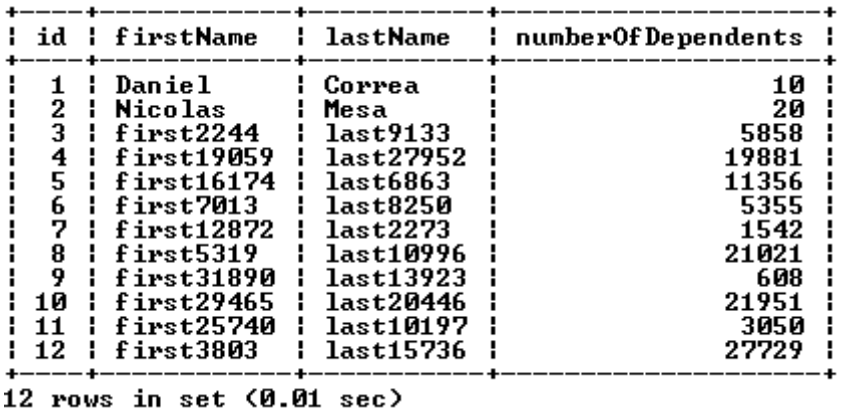
En el código podemos observar las operaciones comunes sobre las bases de datos: inserción, actualización, borrado, selección y búsqueda, a través de los métodos insert, update, delete, fetchAll y find respectivamente.

Inicialmente se tenían dos usuarios creados:

```
+----+-----------+----------+--------------------+
| id | firstName | lastName | numberOfDependents |
+----+-----------+----------+--------------------+
|  1 | Daniel    | Correa   |                 10 |
|  2 | Nicolas   | Mesa     |                 20 |
+----+-----------+----------+--------------------+
2 rows in set (0.00 sec)
```

Figura 19: Estado inicial de la tabla Person

Durante las diez iteraciones, se insertaron datos de forma aleatoria:

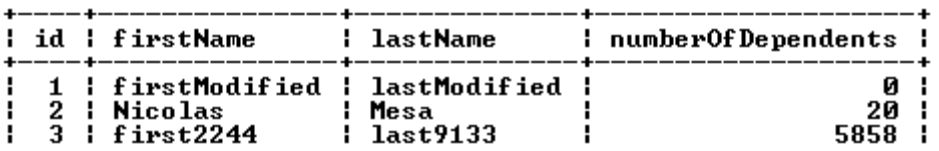


id	firstName	lastName	numberOfDependents
1	Daniel	Correa	10
2	Nicolas	Mesa	20
3	first2244	last9133	5858
4	first19059	last27952	19881
5	first16174	last6863	11356
6	first7013	last8250	5355
7	first12872	last2273	1542
8	first5319	last10996	21021
9	first31890	last13923	608
10	first29465	last20446	21951
11	first25740	last10197	3050
12	first3803	last15736	27729

12 rows in set (0.01 sec)

Figura 20: Inserción de datos aleatorios en la tabla Person

En la siguiente operación, se actualizaron los datos para la persona con el ID 1:



id	firstName	lastName	numberOfDependents
1	firstModified	lastModified	0
2	Nicolas	Mesa	20
3	first2244	last9133	5858

Figura 21: Actualización de los datos para la persona con ID igual a 1

La siguiente operación, fue el borrado de la persona con el ID 2:

id	firstName	lastName	numberOfDependents
1	firstModified	lastModified	0
3	first2244	last9133	5858
4	first19059	last27952	19881

Figura 22: Borrado de la persona con ID igual a 2

Finalmente, a través de las operaciones de selección y búsqueda, encontramos los datos pertenecientes a las personas con ID 5 y 10:

```
5: first16174 last6863
10: first29465 last20446
```

Figura 23: Búsqueda de datos de personas con ID igual a 5 y 10

5.7.6 Análisis de resultados

Los resultados son concluyentes, las operaciones fueron ejecutadas exitosamente sin tener que implementar una sola línea que mezclara la lógica de la aplicación con SQL.

Este patrón junto con el patrón de diseño Adapter le permite al desarrollador construir aplicaciones altamente escalables, al realizar cambios de motor, el Adapter se encarga de que las operaciones comunes sean traducidas correctamente a la sintaxis soportada por el nuevo motor, el patrón de Table Data Gateway simplemente permite crear la interfaz común entre la aplicación y la base de datos.

6. NORMAS DE CODIFICACIÓN PARA LA GENERACIÓN DE CÓDIGO MANTENIBLE

6.1 FORMATO DE ARCHIVOS .PHP

6.1.1 Sangría

Se debe usar 4 espacios para la sangría y nunca usar tabulación. Esto con el fin de que se vea de la misma manera en cualquier entorno de desarrollo sin importar como estén configuradas las tabulaciones.

6.1.2 Longitud máxima de una línea de código

El máximo número de caracteres por línea de código en PHP es de 120, sin embargo, un programador de Zend Framework debe intentar que todas sus líneas de código estén por debajo de 80 caracteres.

6.2 CONVENCIONES DE NOMBRES

6.2.1 Clases

Zend Framework estandarizó los nombres de las clases en las que cada sub-guión indica un directorio. Por ejemplo la clase "Zend_Validate_Regex" está ubicada en la ruta "Zend/Validate/Regex.php". De esta forma el framework sabe dónde está el archivo que contiene la clase que se desea utilizar.

Los nombres de las clases solo deben tener caracteres alfanuméricos con el sub-guión como separador de directorios.

La primera letra de cada palabra debe ser en mayúscula pero nunca debe haber dos mayúsculas seguidas. Por ejemplo la clase "Model_MyHTML" no se debe usar, el nombre correcto es "Model_MyHtml".

6.2.2 Clases abstractas

Las clases abstractas siguen la misma convención de nombres de las clases, la única diferencia es que se les debe agregar la palabra "Abstract" al final. Por ejemplo "Model_MyHtmlAbstract".

6.2.3 Interfaces

Las interfaces se deben nombrar igual que una clase abstracta pero en lugar de poner "Abstract" al final del nombre, se pone la palabra "Interface". Por ejemplo "Model_CarInterface".

6.2.4 Nombres de archivos

Cualquier archivo que contenga código PHP debe terminar con la extensión ".php" a excepción de las vistas que terminan con la extensión ".phtml".

El nombre de un archivo debe estar en la ruta que indica el nombre de la clase y tener el nombre de la última palabra que está en la ruta. Por ejemplo el archivo "Zend_Controller_Front" debe estar en la ruta "Zend/Controller/" y debe ser nombrado "Front.php".

6.2.5 Funciones y métodos

Los nombres de los métodos solo pueden contener valores alfanuméricos. Aunque los números son permitidos solo se deben utilizar cuando sea estrictamente necesario. Por ejemplo una función que convierta una cadena a base64 puede llamarse "base64Encode" sin problema.

Se debe tratar de usar verbos en las funciones para saber qué hacen sin tener que leer su documentación.

Las funciones para acceder a variables privadas o protegidas de una clase siempre deben empezar por "set" para darles un valor y "get" para obtener su valor. Por ejemplo:

```
<?php
class Model_Person
{
    private $_name;

    public function setName($name)
    {
        $this->_name = $name;
    }

    public function getName()
    {
        return $this->_name;
    }
}
```

Se debe hacer lo posible por no usar funciones globales. Es mejor considerar meterlas en una clase estática.

6.2.6 Variables

Los nombres de variables solo pueden contener valores alfanuméricos y solo deben tener sub-guiones si son variables privadas o protegidas de una clase. Se debe evitar usar números a no ser que sea necesario.

Todas las variables privadas o protegidas de una clase deben empezar con sub-guión pero ninguna pública lo debe hacer.

Las primera letra de una variable debe ser minúscula y luego cada palabra debe empezar por mayúscula. Por ejemplo:

```
private $_itemPrice;
```

Las variables deben tener nombres que describan lo que van a almacenar. Las variables $i, $j, y $n solo deben ser usadas en ciclos muy pequeños. En los ciclos que tengan más de 20 líneas de código por dentro, los nombres de estas variables deben ser más descriptivos.

6.2.7 Constantes

Los nombres de las constantes pueden contener valores alfanuméricos y sub-guiones.

El nombre entero de una constante debe estar en mayúsculas y las palabras deben ser separadas por sub-guiones. Por ejemplo:

```
MAX_PRICE = 1000000;
```

6.3 ESTILO DE CODIFICACIÓN

6.3.1 Demarcación de código PHP

El código PHP siempre debe estar entre los demarcadores largos de PHP. Por ejemplo:

```
<?php
    echo "Texto entre demarcadores largos de PHP";
?>
```

En archivos que solo contengan código PHP el demarcador para cerrar el código nunca se debe usar. Por ejemplo:

```
<?php
class Model_Car
{
```

```
}
```

Como se puede ver, el demarcador para cerrar el código "?>" no fue utilizado. Ésto se hace para que no salgan errores si se deja un espacio en blanco después de cerrar el código ya que este tipo de error es muy difícil de encontrar.

6.3.2 Cadenas

Literales

Cuando una cadena es literal (se quiere almacenar o mostrar el valor literal) se debe usar comillas simples ('). Por ejemplo:

```
echo 'Mostrando una cadena literal';
```

Sustitución de variables

En PHP, es posible insertar una variable en una cadena sin tener que hacer concatenación. Para hacer ésto, la cadena tiene que estar entre comillas dobles ("). Por ejemplo:

```
$clientName = 'Camilo';

//opción 1
$sql = "insert into clients(name) VALUES('$clientName')";

//opción 2
$sql = "insert into clients(name) VALUES('{$clientName}')";
```

Concatenación de cadenas

La concatenación siempre se hace con el operador punto "." y se debe poner un espacio antes y otro después del punto. Por ejemplo:

```
$clientName = 'Camilo';

echo 'Nombre del cliente: ' . $clientName;
```

Cuando se está concatenando, se sugiere dividir la instrucción en varias líneas para que sea más fácil de leer alineando el punto (.) con el igual (=). Por ejemplo:

```
$output = 'Aquí se muestra una cadena '
        . 'muy larga por lo que se va '
        . 'dividir en varias líneas.'
```

6.3.3 Vectores

Vectores con índices numéricos

No se permiten números negativos como índices para un vector.

Un vector puede iniciar en cualquier índice positivo, sin embargo se recomienda que empiece desde 0.

Cuando se crea un vector con el constructor, se debe dejar un espacio entra la coma y el siguiente parámetro. Por ejemplo:

```
$colors = array('red', 'blue', 'yellow', 'green');
```

También se puede declarar vectores con varias líneas pero se debe asegurar que el principio de cada línea esté alineado con el anterior. Por ejemplo:

```
//opción 1
$colors = array('red', 'blue', 'yellow',
                'green', 'orange', 'purple');

//opción 2
$colors = array(
    'red', 'blue', 'yellow',
    'green', 'orange', 'purple',
);
```

Si se decide usar la opción 2, se recomienda dejar la coma (,) que se puso al final, de esta forma es más fácil agregar más valores al vector evitando errores por falta de comas (,).

Vectores asociativos

En vectores asociativos, se recomienda dividir la definición del vector en varias líneas para que sea más fácil de leer. Por ejemplo:

```
// opción 1
$person = array('name' => 'Camilo',
    'lastName' => 'Cadavid',
    'nickname' => 'camcad');

//opción 2
$person = array(
    'name' => 'Camilo',
    'lastName' => 'Cadavid',
    'nickname' => 'camcad',
);
```

Si se decide usar la opción 2, se recomienda dejar la coma (,) que se puso al final para que sea más fácil agregar valores evitando errores por falta de comas (,).

6.4 CLASES

6.4.1 Declaración

Las clases deben ser nombradas según el estándar de Zend Framework.

La llave siempre debe ir debajo del nombre de la clase.

Solo debe haber una clase por cada archivo.

Ejemplo:

```
class Model_Car
{
    //aquí va el contenido de la clase
}
```

La herencia o implementación se debe declarar en la misma línea cuando sea posible (si no excede el número máximo de caracteres que debe tener una línea). Por ejemplo:

```
class Model_Car extends Model_VehicleAbstract
{

}
```

6.4.2 Variables de clase

Las variables de clase deben ser nombradas según el estándar de Zend Framework.

Todas las variables de clase deben estar al principio de la clase por encima de todos los métodos. Por ejemplo:

```
class Model_Car
{
    private $_color;

    public function setColor($color)
    {
        $this->_color = $color;
    }

    public function getColor($color)
    {
```

```
        return $this->_color;
    }
}
```

Se debe especificar la visibilidad de cada variable de clase declarándolas como privadas (private), protegidas (protected) o públicas (public).

El uso de variables de clase públicas está permitido pero debe ser evitado. Es mejor usar los métodos "set" y "get".

6.5 FUNCIONES Y MÉTODOS

6.5.1 Declaración

Las funciones y métodos deben ser declaradas de acuerdo con el estándar de Zend Framework.

Los métodos en una clase siempre deben especificar su visibilidad: privada (private), protegida (protected) o pública (public).

La llave debe estar ubicada debajo del nombre, y no es permitido tener espacios entre el nombre y el primer paréntesis de la función.

Ejemplo:

```
class Model_Client
{
    public function sendEmail($text)
    {
        //las instrucciones para enviar el correo van aquí
    }
}
```

Si el número de parámetros para la función excede la longitud máxima de la línea, los parámetros pueden dividirse en dos o más líneas. En este caso el paréntesis de cierre debe estar en su propia línea con la llave que abre la función. Ésto con el fin de que se fácil agregar más parámetros cuando se necesite. Por ejemplo:

```
class Model_Client
{
    public function __construct($name, $lastName, $phone,
        $address, $email, observations
    ){
        //las instrucciones para el constructor van aquí
    }
}
```

Si se desea pasar argumentos por referencia, se debe especificar desde la definición de la función. Nunca se debe pasar argumentos por referencia desde donde se llama la función. Por ejemplo:

```
class Model_Security
{
    public static function setRandomPassword(&$client){
        /* las instrucciones para generar la contraseña y
ponérsela al
         * cliente van aquí */
}
```

El valor de retorno no debe estar entre paréntesis ya que hace que el código sea más difícil de leer. Por ejemplo:

```
class Model_Client
{
    private $_name;

    public function getName()
    {
        return $this->_name;
    }
}
```

6.5.2 Uso

Los parámetros que se le pasan a una función deben estar separados por un espacio después de cada coma (,). Por ejemplo:

```
echo $mathObject->add(10, 20, 30);
```

6.6 INSTRUCCIONES DE CONTROL

6.6.1 Instrucciones tipo If, Elseif, Else

Todas las instrucciones de tipo “if” o “elseif” deben tener un espacio antes del paréntesis que abre y otro después del paréntesis que cierra.

La llave que abre el if debe estar en la misma línea que el condicional, y la llave que lo cierra debe estar en su propia línea.

Por ejemplo:

```
if ($count > 10) {
     echo $Count;
```

```
}
```

Si la longitud de los condicionales excede la longitud máxima de la línea, se debe partir los condicionales en varias líneas alineando los operadores. En este caso, el paréntesis que cierra el if debe quedar en una línea aparte junto con la llave que abre. Ésto para que sea más fácil agregar o quitar condiciones. Por ejemplo:

```
if (($count > 10)
    && ($count < 100)
    || ($count === 1000)
) {
    echo $count;
}
```

El uso del elseif es exactamente igual al if. Por ejemplo:

```
if (($count > 0)
    && ($count < 100)
) {
    echo 'Too little';
} elseif (($count > 100)
          && ($count < 1000)
) {
    echo 'Normal'
} else{
    echo 'Value out of range';
}
```

6.6.2 Instrucciones de tipo switch

Las instrucciones de tipo switch deben tener un espacio antes del paréntesis que abre y otro después del paréntesis que cierra.

El “default” nunca debe ser omitido en un switch.

Por ejemplo:

```
switch ($programmingLanguage) {
    case 'java':
        break;
    case 'php':
        echo 'PHP';
        break;
    default:
        echo 'Unknown language';
        break;
}
```

Algunas veces se decide no poner break para que el switch se siga ejecutando. Si ésto ocurre, se debe poner un comentario que lo diga para que nadie piense que es un error y decida “corregirlo”.

6.7 DOCUMENTACIÓN

Uno de los aspectos más importantes para generar código entendible es el uso adecuado de la documentación. Existen etiquetas que pueden ser usadas a través de las funciones y métodos de los archivos para la generación automática de documentación, la cual puede ser utilizada por terceros para la comprensión del código.

6.7.1 Archivos

Todos los archivos que tengan código PHP deben tener las etiquetas: category, package, copyright, license, y link.

6.7.2 Clases

Todas las clases deben tener las siguientes etiquetas: Category, package, copyright, license, version, link, y deprecated.

6.7.3 Funciones

Todas las funciones deben tener las siguientes etiquetas: param, return, y throws.

6.7.4 Etiquetas

A continuación se explican las etiquetas listadas anteriormente:

@category: “Se usa para denotar agrupaciones de paquetes (@package)” (Beaver). Sin embargo, desde que la etiqueta @package comenzó a soportar subdivisiones, esta etiqueta entró en desuso. Por ejemplo:

```
/**
 * @category Usuarios
 */
```

@copyright: “Se usa para informar sobre los derechos sobre la clase o archivo” (Beaver G.). Se recomienda incluir los años en los que los derechos son válidos y la organización que los posee. Por ejemplo:

```
/**
 * @copyright 2010-2013 Mi Empresa
 */
```

@deprecated: “Se usar para indicar que el bloque documentado no se debe usar y que en una futura versión se va a eliminar” (Beaver G. , Deprecated). Es importante incluir la versión desde que entró en desuso y si se desea una pequeña descripción de la razón o de cual función o clase se debe usar en su lugar. Por ejemplo:

```
/**
 * @deprecated 1.9.1 Esta función muy lenta. Mejor usar buildTree
 */
```

@license: “Se usa para indicar la licencia que le aplica al bloque documentado” (Beaver G. , License). Se debe incluir la url de la licencia con su respectivo nombre. Además solo se debe tener una licencia por archivo para no generar confusiones. Por ejemplo:

```
/**
 * @license http://opensource.org/licenses/mit-license.php MIT
 */
```

@link: “Se usa para indicar una relación entre el bloque documentado y una página web” (Beaver G. , Link). Se debe incluir la url y una descripción de la relación entre el bloque y la página web. Por ejemplo:

```
/**
 * @link http://www.ejemplo.com/program/build-tree.html
Documentación
 * crear el árbol
 */
```

@package: “Se usa para categorizar el bloque documentado en subdivisiones” (Beaver G. , Package). Por ejemplo:

```
/**
 * @package \transporte\vehiculo\carro
 */
```

@param: “Se usa para documentar un parámetro que se le pasa la función documentada” (Beaver G. , Param). Debe tener el tipo de dato seguido por su nombre y la descripción. Por ejemplo:

```
/**
 * @param Model_Car $car El carro a ser matriculado
 */
```

@return: “Se usa para definir el valor de salida de la función” (Beaver G. , Return). Se debe incluir el tipo de dato a retornar y se le debe poner una corta descripción. Por ejemplo:

```
/**
 * @return Model_Car|NULL Si se encuentra el carro se retorna, de
lo
 * contrario se retorna NULL
 */
```

@throws: “Se usa para indicar que si hay un error lanza una excepción” (Teutsch). Por ejemplo

```
/**
 * @throws Exception Si no se pudo conectar a la base de datos
 */
```

@version: “Se usa para documentar la versión del bloque documentado” (Beaver G. , Version). Por ejemplo:

```
/**
 * @version Version 3.3.10
 */
```

7. PRUEBA DEL NIVEL DE ENTENDIBILIDAD DE UNA APLICACIÓN

Para evaluar el nivel de entendibilidad de una sección de código se propuso crear una aplicación con dos métodos que tuvieran el mismo comportamiento. Dichos métodos tendrían una implementación diferente, uno siguiendo buenas prácticas de programación y otro no. Dichos métodos pueden ser encontrados en el Anexo 1 y 2.

Para evaluar el nivel de entendibilidad, se midió el tiempo que la persona se demoraba en realizar tres cambios funcionales sobre ambos métodos.

El funcionamiento de la aplicación era:

1. Consumir un servicio web a través de SOAP. Dicho servicio provee una interfaz para consultar las ciudades de cada país que el servicio soporta para consultar el clima, y la consulta del clima de una ciudad específica.

2. Obtener las ciudades soportadas por el servicio para Colombia.

3. Generar por cada ciudad una población hipotética.

4. Obtener las diez ciudades con más población.

5. Para cada ciudad, obtener el estado climático a través del servicio web.

6. Mostrar la ciudad, la población y la temperatura, para las ciudades con temperatura mayor a 70 °F.

Los cambios funcionales en la aplicación pedidos eran:

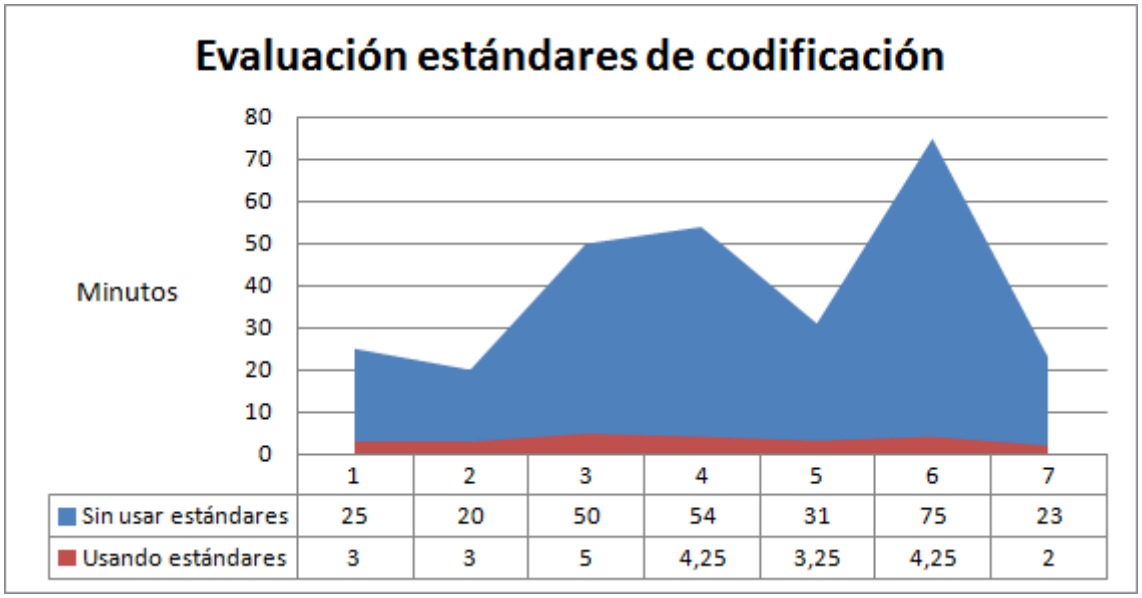
1. Obtener las diez ciudades con menos población.

2. Mostrar el nombre del aeropuerto.

3. Mostrar la temperatura en grados Celsius.

7.1 RESULTADOS

En la siguiente gráfica se puede apreciar los resultados de la evaluación. Debido a la complejidad del problema, solo fue posible evaluar las prácticas estándares de codificación con siete personas:



	1	2	3	4	5	6	7
Sin usar estándares	25	20	50	54	31	75	23
Usando estándares	3	3	5	4,25	3,25	4,25	2

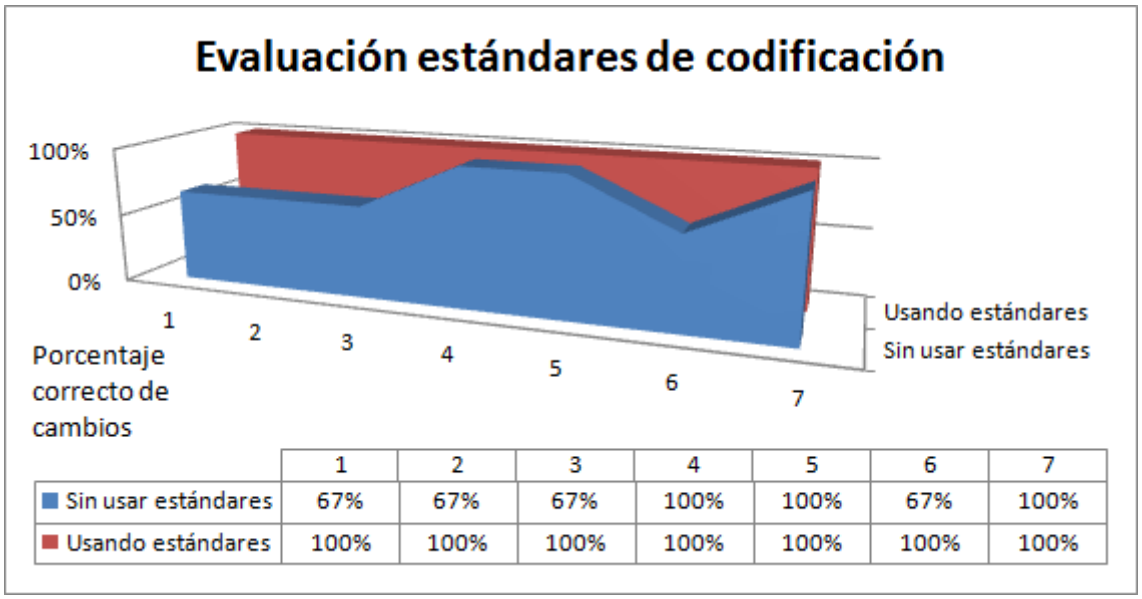
Figura 24: Resultado de la evaluación de los estándares de codificación

Aunque la muestra poblacional no es significativa, podemos ver que la tendencia del tiempo sobre el cambio en el método que no usaba estándares de codificación era alta, mientras que la tendencia del tiempo para el método usando estándares de codificación era baja.

En promedio, las siete personas de demoraron en corregir el método que no usaba estándares de codificación 40 minutos, mientras que en el método con uso de estándares fue de 4 minutos.

El nivel de entendibilidad está estrechamente relacionado con el uso de estándares de codificación: a mayor uso de estándares mayor entendibilidad, y a menor uso de estándares menor entendibilidad.

Se usó además la métrica de porcentaje correcto de cambios, la cual es el porcentaje del número de cambios correctos sobre el total de cambios por realizar. En la siguiente gráfica se puede apreciar el resultado:



	1	2	3	4	5	6	7
Sin usar estándares	67%	67%	67%	100%	100%	67%	100%
Usando estándares	100%	100%	100%	100%	100%	100%	100%

Figura 25: Porcentaje correcto de cambio

Para el método sin uso de estándares de codificación, se detectó además, que el promedio de porcentaje correcto de cambios fue del 80,95%, mientras que en el método con uso de estándares fue del 100%.

Un análisis posterior sobre los cambios realizados por las personas, demostró que para el método que no usaba estándares de codificación, la persona prefirió programar partes enteras de la funcionalidad, en vez de tratar de entender su funcionamiento, mientras que los cambios sobre el buen uso de estándares fue muy similar, esto es, cambios lineales y poco drásticos.

Algunas de las opiniones sobre cada método del ejercicio pueden ser vistas a continuación:

Método sin uso de estándares de codificación

- Es una tontería.
- Es más fácil reescribirlo.
- Quien es el desarrollador?
- Poco portable.
- Old school, no aprovecha el poder de PHP.
- Mala implementación.

- Código sin comentar.
- Recuerda un código en C, escrito por alguien que aprendió a programar empíricamente.
- Es más fácil rehacer todo el problema.
- Muy enredado.
- Expresión regular muy complicada.
- Intentan confundirte usando código innecesario.

Método con uso de estándares de codificación

- Muy fácil de modificar.
- La documentación ayuda bastante.
- Es un código más limpio.
- Es más fácil entender que hace cada línea de código.
- Funciones apropiadas para resolver un problema.
- Mucho más fácil de modificar y entender.
- Es más fácil entender el código, por las buenas prácticas de programación.
- Mucho comentario.

8. CONCLUSIONES Y CONSIDERACIONES FINALES

- Los patrones de diseño ayudan a generar código mantenible, ya que hacen que un cambio sea fácil de hacer y normalmente solo hay que hacerlo en una parte del código. De esta forma se evita tener que buscar por todo el código donde hacer el cambio y como solo hay un cambio, éste es menos propenso a errores.

- Los patrones de diseño ayudan a generar código escalable. Ésto se pudo ver fácilmente con el patrón Adapter. Si una compañía desea cambiar de motor de base de datos porque necesita una más potente, no hay problema ya que el adaptador se encarga de "traducir" las sentencias SQL al lenguaje del nuevo motor de base de datos.

- Los patrones de diseño ayudan a generar código entendible debido a que son conocidos por muchos programadores. Si a un programador se le indica el patrón utilizado, lo más seguro es que sepa de cual patrón se está hablando.

- Siguiendo el estándar de codificación de Zend, se mejora notablemente la entendibilidad y mantenibilidad del código.

- El nombre de las variables es extremadamente importante pues describen el valor que están guardando y esto resulta muy útil para alguien que esté tratando de entender el código.

- La documentación del código es vital para que los desarrolladores entiendan lo que hace el código y como lo hace. De esta forma se puede hacer cambios de una manera más simple y rápida.

- Es mejor escribir el código bien desde el principio y no tener que escribirlo otra vez cuando se vaya a hacer un cambio porque no se pudo entender qué hacía.

- Los cambios en la aplicación de código mal escrito es 11 veces más en promedio que los cambios en la aplicación que sigue los estándares. Teniendo esto en cuenta, el costo de mantener una aplicación que tiene un código mal escrito puede llegar a ser 11 veces más grande que el de una aplicación que sigue los estándares de codificación.

9. BIBLIOGRAFÍA

Beaver, G. (s.f.). *Category*. Recuperado el 20 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.category.pkg.html

Beaver, G. (s.f.). *Copryright*. Recuperado el 20 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.copyright.pkg.html

Beaver, G. (s.f.). *Deprecated*. Recuperado el 21 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.deprecated.pkg.html

Beaver, G. (s.f.). *License*. Recuperado el 21 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.license.pkg.html

Beaver, G. (s.f.). *Link*. Recuperado el 21 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.link.pkg.html

Beaver, G. (s.f.). *Package*. Recuperado el 21 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.package.pkg.html

Beaver, G. (s.f.). *Param*. Recuperado el 21 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.param.pkg.html

Beaver, G. (s.f.). *Return*. Recuperado el 21 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.return.pkg.html

Beaver, G. (s.f.). *Version*. Recuperado el 23 de 09 de 2012, de phpDocumentor: http://manual.phpdoc.org/HTMLSmartyConverter/HandS/phpDocumentor/tutorial_tags.version.pkg.html

Christopher, A., Ishikawa, S., MurraySilverstein, Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language.* New York: Oxford University.

DaveOrme. (20111 de 17 de Marzo). *History Of Patterns*. Recuperado el 2012 de Marzo de 10, de C2: http://c2.com/cgi-bin/wiki?HistoryOfPatterns

Fowler, M. (1999). *Refactoring. Improving the Design of Existing Code.* Addison-Wesley.

Framework, Z. (s.f.). *Coding Style*. Recuperado el 2012 de 04 de 23, de Coding Style: http://framework.zend.com/manual/en/coding-standard.coding-style.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* Illinois.

Iafrancesco, G. (2003). *La investigación es educación y pedagogía.* Bogotá: Magisterio.

Kerievsky, J. (2004). *Refactoring To Patterns.* Addison-Wesley.

Martin, F. (s.f.). *Refactorings in Alphabetic Order*. Obtenido de http://refactoring.com/catalog/index.html

Tamayo, M. T. (1999). *Serie: Aprender a investigar módulo 5: el proyecto de investigación.* Bogotá: ICFES.

Técnicas avanzadas de programación. (s.f.). *Refactoring*. Recuperado el 04 de 05 de 2012, de Técnicas avanzadas de programación: http://utntadp.com.ar/material/refactoring

Teutsch, M. (s.f.). *Introduction to phpDoc*. Recuperado el 23 de 09 de 2012, de phpMaster: http://phpmaster.com/introduction-to-phpdoc/

Digitalus. (s.f.). *CMS - Digitalus Media*. Recuperado el 14 de 10 de 2012, de Digitalus: http://digitalusmedia.com/cms

Fowler, M. (s.f.). *Table Data Gateway*. Recuperado el 15 de 09 de 2012, de Martin Fowler: http://www.martinfowler.com/eaaCatalog/tableDataGateway.html

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., & Stafford, R. (2002). *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns.* Upper Saddle River: Addison-Wesley.

Wikipedia. (13 de Junio de 2012). *Mantenibilidad*. Recuperado el 3 de Julio de 2012, de Wikipedia: http://es.wikipedia.org/wiki/Mantenibilidad

Zend Developer Zone. (5 de Mayo de 2008). *Decorators with Zend_Form*. Recuperado el 18 de Agosto de 2012, de Zend Developer Zone: http://devzone.zend.com/1240/decorators-with-zend_form/

Zend, T. (s.f.). *Zend Db Table*. Recuperado el 16 de 09 de 2012, de Zend Framework: http://framework.zend.com/manual/1.12/en/zend.db.table.html

HYPERLINK "http://en.wikipedia.org/wiki/William_Opdyke" \o "William Opdyke" Opdyke, William F. ; Johnson, Ralph E. (September 1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems". *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)*. ACM.

Mens, Tom and Tourwé, Tom (2004) A Survey of Software Refactoring, IEEE Transactions on Software Engineering, February 2004 (vol. 30 no. 2), pp. 126–139

ANEXO 1: BAD METHOD

```
public function badAction() {
    $soap = new
Zend_Soap_Client_DotNet('http://www.webservicex.net/globalweather.
asmx?wsdl');
    $data = $soap->GetCitiesByCountry(array('CountryName' =>
'colombia'));

    $dataSet = (array) simplexml_load_string($data);

    $cities = array();
    foreach ($dataSet['Table'] as $algo) {
        foreach ($algo as $key => $value) {
            if ($key === 'City') {
                $name = '';
                $cut = false;
                $val = (string) $value;
                for ($x = 0; $x < strlen((string) $value); $x++) {
                    if ($val[$x] !== '/') {
                        $name .= $val[$x];
                    } else {
                        $cut = true;
                        break;
                    }
                }

                if ($cut) {
                    $temp = '';
                    for ($y = 0; $y < strlen($name) - 1; $y++) {
                        $temp .= $name[$y];
                    }
                    $name = $temp;
                }

                $cities[$name] = mt_rand(100000, 5000000);
            }
        }
    }


    $unordered = array();
    foreach ($cities as $index) {
        $unordered[] = $index;
```

```
    }


    for ($a = 0; $a < count($unordered); $a++) {
        for ($b = 0; $b < count($unordered) - 1; $b++) {
            if ($unordered[$b] < $unordered[$b + 1]) {
                $aux = $unordered[$b];
                $unordered[$b] = $unordered[$b + 1];
                $unordered[$b + 1] = $aux;
            }
        }
    }

    $arr = array_flip($cities);

    $final = array();
    for ($c = 0; $c < 9; $c++) {
        $final[$arr[$unordered[$c]]]['population'] =
$unordered[$c];
    }

    foreach ($final as $key => $fin) {
        $weather = $soap->GetWeather(array('CountryName' =>
'colombia', 'CityName' => $key));

        if (preg_match('/<Temperature>\s*(?<F>(?:[7-9][0-
9]|\d{3,}))[^<]+<\/Temperature>/i', $weather, $matches)) {
            $final[$key]['temperature'] = $matches['F'];
        }
    }

    $this->view->items = $final;
}
```

ANEXO 2: GOOD METHOD

```
public function goodAction() {
    // Obtenemos la descripción del servicio
    $soap = new
Zend_Soap_Client_DotNet('http://www.webservicex.net/globalweather.
asmx?wsdl');

    /*
     * Llamamos al método GetCitiesByCountry el cual obtiene la
lista de ciudades de Colombia
     * que tienen soporte para la obtención del clima.
     */
    $data = $soap->GetCitiesByCountry(array('CountryName' =>
'Colombia'));

    // Procesamos la salida del servicio
    $dataSet = simplexml_load_string($data);

    // El array de ciudades contiene las ciudades soportadas
    $cities = array();

    // Se recorre cada tabla
    foreach ($dataSet->Table as $table) {
        // La ciudad se compone del nombre y el aeropuerto del
cual se obtiene el estado climático
        $city = (string) $table->City;

        // Separamos los datos
        $data = explode('/', $city);
        $name = trim($data[0]);

        if (isset($data[1])) {
            $airport = trim($data[1]);
        }

        // Se genera aleatoriamente la población de cada ciudad
        $cities[$name]['population'] = mt_rand(100000, 5000000);
    }

    // Llevamos la población a un array el cual es usado para el
ordenamiento
    $population = array();
    foreach ($cities as $name => $values) {
```

```
        $population[] = $cities[$name]['population'];
    }

    // Se ordena el array de ciudades de forma descendente basados
en la población
    array_multisort($population, SORT_DESC, $cities);

    // Se obtienen las 10 ciudades con más población
    $highest_population = array_slice($cities, 0, 10, true);

    // Recorremos cada ciudad
    foreach ($highest_population as $name => $values) {
        // A través del método GetWeather obtenemos el estado
climático para la ciudad
        $data = $soap->GetWeather(array('CountryName' =>
'Colombia', 'CityName' => $name));

        // Se convierte la entrada a UTF-16, para el correcto
procesamiento
        $data = mb_convert_encoding($data, 'UTF-16');

        // Procesamos la salida del servicio
        $weather = simplexml_load_string($data);

        // Se obtiene la temperatura
        $temperature = (string) $weather->Temperature;

        // Obtenemos los componentes de Fahrenheit y Celsius
        $found =
        preg_match('/\s*                # Espacios en blanco al
inicio de la cadena
                    (?<Fahrenheit>      # Llevamos al índice
Fahrenheit la temperatura
                    [7-9][0-9]|\d{3,}   # La temperatura que
escogemos es mayor o igual a 70°F
                    )
                    .*                  # Descarta caracteres luego
de la temperatura
                    \(                  # Dentro de paréntesis se
encuentra la temperatura en Celsius
                    (?<Celsius>         # Llevamos al índice
Celsius la temperatura
                    [1-9][0-9]          # La temperatura que
escogemos es mayor oigual a 10°C
                    )
                    .*
                    \)
```

```
                    \s*                 # Espacios al final de la
cadena
                    /x', $weather->Temperature, $matches);

        if ($found) {
            // Si es encontrada la temperatura se agrega a la
ciudad
            $highest_population[$name]['temperature'] =
$matches['Fahrenheit'];
        }
    }

    $this->view->items = $highest_population;
}
```

ESCUELA DE INGENIERÍA DE ANTIOQUIA

ACTA DE EVALUACIÓN FINAL DE TRABAJO DE GRADO

Fecha: (dd/mm/aa)	22/11/2012
Nombre del proyecto:	**Buenas prácticas para el desarrollo de aplicaciones web a través de patrones de diseño usando el marco de trabajo Zend Framework**
Director del proyecto:	Santiago Villegas Giraldo

Nombre del estudiante	**Programa académico**
Daniel Guillermo Correa Isaza	Ingeniería Informática
Nicolás Mesa Fernández	Ingeniería Informática

Nombre del Jurado:	

Evaluación del proyecto: Espacio exclusivo para jurado

___No aprobado _√_Aprobado sin mención

___con Mención Pública ___ con Mención honorífica ___Trabajo laureado

Justificación del reconocimiento: (Artículo 28 del Acuerdo 11: "El director del Programa presentará el acta final de evaluación al Consejo Académico, donde consta la solicitud de mención especial debidamente justificada y el **Consejo determinará** si se otorga o no"). La justificación debe tener mínimo 500 palabras.

CARLOS JAIME NORENA MEJIA
Director del Programa

Santiago Villegas G.
Director del Trabajo de Grado

Jurado (Si lo hubo)

Jurado (Si lo hubo